University of Colorado - Boulder

ECEN 2370
Embedded Systems — Fall 2024

Final Project - Tetris on STM32F429i

Sam Walker Xavion Cowans

Sunday, December 15, 2024

Project Description

The objective of this project is to design and implement a functional Tetris game on the STM32F429i development board, showcasing skills in embedded systems programming, peripheral integration, and real-time scheduling. My implementation of the game consists of three main screen states:

- **Main Menu Screen:** Displays a "Start" button and a randomized arrangement of the seven Tetris blocks.
- **Game Screen:** Allows the player to interact with falling Tetris blocks using the touch screen and blue user button. Blocks fall at a rate of 4 squares per second, and the gameplay timer is displayed in the top right corner. The next block to drop is also shown below the timer.
- **Game Over Screen:** Activates when a block reaches the top row. This screen displays the total time survived, along with a breakdown of the number of singles, doubles, triples, and tetrises cleared.

Key features include randomized block generation using the RNG peripheral, block rotation with the user button, left and right movement via touch screen, and row-clearing logic for single or consecutive lines. The game leverages timers for block falling rates and elapsed time tracking.

Project Scope and Timeline

Scope

The project involves using the following peripherals and features:

- RNG for randomizing the Tetris block generation.
- Timer 2 to display real-time elapsed time during gameplay.
- Timer 5 to control the falling rate of the Tetris blocks.
- The LCD touch screen for displaying the game and user interaction.
- The blue user button to rotate blocks.

Timeline

- **11/15**: Submit project proposal.
- **11/19**: Deliver project scope, timeline, and testing strategy.
- **12/03**: Project check-in to demonstrate progress.
- **12/11**: Submit code for review.
- **12/12**: Demo and interview grading.
- **12/15**: Submit lab write-up and documentation.

Project Documentation

The project documentation includes the work breakdown, testing strategy, and use cases.

Work Breakdown

- **System Initialization:** Configure peripherals (RNG, timers, touch screen, GPIO) for block generation, timing, and input.
- **Main Menu:** Display a "Start" button and Tetris blocks; handle touch input to start the game.
- **Game Screen:**
  - Generate blocks with RNG and manage movement (touch) and rotation (button).
  - Control block falling and row clearing with Timer 5.
- **Game Over Screen:** Trigger on game-over conditions, display statistics, and allow return to the main menu.

Testing Strategy

- **Peripheral Testing:** Confirm RNG, timers, and touch screen function as expected.
- **Functional Testing:** Validate movement, collision detection, row clearing, and screen transitions.
- **System Testing:** Test integration, edge cases, and overall stability.
- **Debugging:** Fix errors and verify gameplay statistics.

Use Cases

- Players interact with blocks to clear rows and score points.
- The game updates gameplay statistics and transitions smoothly between screens.
- The game-over screen displays results (time, rows cleared, etc.).

Code Documentation

The Tetris game implementation is structured across four core modules: `ApplicationCode.c`, `Game.c`, `GameOver.c`, and `MainMenu.c`. Each module contains functions designed to manage specific aspects of the game's functionality. Below is a summary of the key functions.

Application Code

`ApplicationCode.c` initializes the application and manages state transitions. (See Appendix A.1)

- `ApplicationInit`: Configures the LCD, RNG, and touch functionalities. Initializes the monitor for debugging.
- `MainMenu`: Displays the main menu screen.
- `Game`: Handles the game loop, updating the screen and managing block behavior.
- `EXTI15_10_IRQHandler`: Processes touch screen interrupts to handle user input and transitions between game states.
- `HAL_TIM_PeriodElapsedCallback`: Manages timer interrupts for time tracking and block movement.

Game Logic

`Game.c` implements the core Tetris gameplay mechanics. (See Appendix A.2)

- `InitializeGame`: Sets up game variables, timers, and the initial block.
- `RenderGameScreen`: Draws the Tetris grid, static blocks, and falling block on the screen.
- `UpdateFallingBlock`: Manages the downward movement of the falling block and detects collisions.
- `MoveFallingBlockLeft/Right`: Moves the falling block horizontally if there is no collision.
- `RotateFallingBlock`: Rotates the falling block if it does not overlap with other blocks or boundaries.
- `ClearFullRows`: Detects and clears completed rows, shifting rows above downward.
- `SpawnRandomBlock`: Randomly selects the next block and spawns it at the top of the grid.
- `DisplayTimer`: Displays the elapsed time in minutes and seconds at the top of the screen.

Game Over Handling

`GameOver.c` handles game-over scenarios. (See Appendix A.3)

- `CheckGameOver`: Checks the top row of the grid for any blocks, indicating game over.
- `RenderGameOverScreen`: Displays the game-over screen with statistics such as time survived and cleared rows.

Main Menu Management

`MainMenu.c` displays the main menu screen and handles navigation. (See Appendix A.4)

- `DisplayMenu`: Renders the main menu screen with randomly positioned Tetris blocks and a "Play" button.
- `DrawPlayButton`: Draws the "Play" button in the center of the screen.
- `PlaceBlock`: Randomly positions blocks on the screen, ensuring no overlap with the "Play" button or other blocks.

Struggles and Obstacles

The development of the Tetris game presented several challenges, two of which stood out as particularly significant:

1. Timing Issues

The first major challenge involved managing the timing for the game clock. Initially, I attempted to use a timer interrupt with a prescaler of 83 and an autoreload value of 999999 to generate a 1-second timer interrupt, based on the STM32's clock frequency of 84 MHz. However, the falling blocks did not behave as expected, which led to confusion. After extensive debugging, I discovered that using a `HAL_Delay` function for the falling rate was interfering with the timer setup. By integrating the block falling mechanism into a timer-based setup, I resolved the issue, achieving accurate timing.

2. Complexity of the Rotation Function

Implementing the rotation function for the Tetris blocks was another significant obstacle. The function required handling dense nested loops and specific indexing for different cases, such as detecting collisions and ensuring valid positions after rotation. Debugging was a slow process, as each iteration required testing multiple scenarios before introducing additional logic. This process was time-intensive and made the rest of the functions feel relatively straightforward in comparison.

Takeaways

This project provided valuable lessons in debugging and system design:

- **Timing and Peripheral Interaction:** I learned the importance of understanding how different mechanisms, such as delays and timers, interact in embedded systems. Transitioning to a timer-based approach taught me to rely on interrupt-driven designs for more reliable and predictable timing.
- **Planning Complex Logic:** The challenges with the rotation function highlighted the importance of planning and testing logic thoroughly before implementation. Breaking down the problem into smaller, testable components would have saved time and effort.

Fresh Restart Strategies

If I were to restart this project, I would approach certain aspects differently to save time and improve efficiency:

- **Plan Rotations Ahead of Coding:** I would map out the logic for rotations, including edge cases, on paper or in a flowchart before coding. This would reduce the trial-and-error debugging process and provide a clear framework for implementation.
- **Focus on Modular Design:** Breaking complex functions like rotations into smaller subfunctions would allow for easier testing and debugging of individual components.
- **Consider Timer Logic Early:** I would implement and test the timer-based block falling mechanism at the start of the project to avoid timing inconsistencies caused by mixed approaches like delays and interrupts.

Future Improvements

- **Score Metric:** Implementing a point-based scoring system would make the game more engaging. Points could be awarded based on the type of row cleared (e.g., singles, doubles, triples, tetrises) with higher scores for clearing multiple rows simultaneously.
- **Dynamic Falling Speed:** To increase difficulty and mimic the behavior of classic Tetris, the falling blocks could start at a slower speed and accelerate as the game progresses. This would require adjusting the timer interrupt frequency dynamically based on the game's progression.
- **High Score Tracking:** Adding persistent memory storage to keep track of the highest score would enhance replayability. The high score could be displayed on the main menu screen and updated whenever a new record is achieved.

These improvements would not only enrich the player's experience but also provide additional opportunities to explore embedded systems concepts like dynamic memory handling and timer configuration.

Appendix

A.1 ApplicationCode.c

```c
/*
 * ApplicationCode.c
 *
 *  Created on: Dec 30, 2023 (updated 11/12/2024) Thanks Donavon!
 *      Author: Xavion
 */

#include "ApplicationCode.h"
#include "MainMenu.h"
#include "Game.h"
#include "GameOver.h"

/* Static variables */


extern void initialise_monitor_handles(void);

#if COMPILE_TOUCH_FUNCTIONS == 1
static STMPE811_TouchData StaticTouchData;
#if TOUCH_INTERRUPT_ENABLED == 1
static EXTI_HandleTypeDef LCDTouchIRQ;
void LCDTouchScreenInterruptGPIOInit(void);
#endif // TOUCH_INTERRUPT_ENABLED
#endif // COMPILE_TOUCH_FUNCTIONS

volatile AppState currentState = MAIN_MENU;

void ApplicationInit(void)
{
        initialise_monitor_handles(); // Allows printf functionality
    LTCD__Init();
    LTCD_Layer_Init(0);
    LCD_Clear(0,LCD_COLOR_WHITE);

    // Initialize the random number generator with a unique seed
    uint32_t seed;
    HAL_RNG_GenerateRandomNumber(&hrng, &seed);
    srand(seed);

    #if COMPILE_TOUCH_FUNCTIONS == 1
        InitializeLCDTouch();

        // This is the orientation for the board to be direclty up where the buttons are vertically above the screen
        // Top left would be low x value, high y value. Bottom right would be low x value, low y value.
        StaticTouchData.orientation = STMPE811_Orientation_Portrait_2;

        #if TOUCH_INTERRUPT_ENABLED == 1
        LCDTouchScreenInterruptGPIOInit();
        #endif // TOUCH_INTERRUPT_ENABLED

        #endif // COMPILE_TOUCH_FUNCTIONS
}

void MainMenu(void)
{
        DisplayMenu();
}

void Game(void)
{
        if(currentState == GAME_SCREEN)
```

```
        {
                RenderGameScreen();
                UpdateFallingBlock();
        CheckGameOver();
        }
}

// TouchScreen Interrupt
#if TOUCH_INTERRUPT_ENABLED == 1

void LCDTouchScreenInterruptGPIOInit(void)
{
        GPIO_InitTypeDef LCDConfig = {0};
    LCDConfig.Pin = GPIO_PIN_15;
    LCDConfig.Mode = GPIO_MODE_IT_RISING_FALLING;
    LCDConfig.Pull = GPIO_NOPULL;
    LCDConfig.Speed = GPIO_SPEED_FREQ_HIGH;

    // Clock enable
    __HAL_RCC_GPIOA_CLK_ENABLE();

    // GPIO Init
    HAL_GPIO_Init(GPIOA, &LCDConfig);

    // Interrupt Configuration
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

        LCDTouchIRQ.Line = EXTI_LINE_15;

}

#define TOUCH_DETECTED_IRQ_STATUS_BIT   (1 << 0)  // Touchscreen detected bitmask

void EXTI15_10_IRQHandler()
{
    HAL_NVIC_DisableIRQ(EXTI15_10_IRQn); // Disable the IRQ to avoid re-entrancy
    bool isTouchDetected = false;

    static uint32_t count;
    count = 0;
    while (count == 0) {
        count = STMPE811_Read(STMPE811_FIFO_SIZE);
    }

    // Disable touch interrupt bit on the STMPE811
    uint8_t currentIRQEnables = ReadRegisterFromTouchModule(STMPE811_INT_EN);
    WriteDataToTouchModule(STMPE811_INT_EN, 0x00);

    // Clear the interrupt bit in the STMPE811
    uint8_t statusFlag = ReadRegisterFromTouchModule(STMPE811_INT_STA);
    uint8_t clearIRQData = (statusFlag | TOUCH_DETECTED_IRQ_STATUS_BIT); // Write one to clear bit
    WriteDataToTouchModule(STMPE811_INT_STA, clearIRQData);

    uint8_t ctrlReg = ReadRegisterFromTouchModule(STMPE811_TSC_CTRL);
    if (ctrlReg & 0x80) {
        isTouchDetected = true;
    }

    // Determine if it is pressed or unpressed
    if (isTouchDetected) { // Touch has been detected
        DetermineTouchPosition(&StaticTouchData);

        if (currentState == MAIN_MENU) {
            // Handle touch in the main menu
            currentState = GAME_SCREEN;  // Transition to the game screen
            InitializeGame();            // Initialize game state
            RenderGameScreen();          // Show the game grid
        } else if (currentState == GAME_SCREEN) {
```

```
            // Handle touch during the game
            uint16_t screenMidpoint = LCD_PIXEL_WIDTH / 2;

            //Flipped because i flipped the pixels 180deg
            if (StaticTouchData.x < screenMidpoint) {
                // Left half of the screen
                MoveFallingBlockRight();
            } else {
                // Right half of the screen
                MoveFallingBlockLeft();
            }
        }
    }

    // Reset FIFO
    STMPE811_Write(STMPE811_FIFO_STA, 0x01);
    STMPE811_Write(STMPE811_FIFO_STA, 0x00);

    // Re-enable IRQs
    WriteDataToTouchModule(STMPE811_INT_EN, currentIRQEnables);
    HAL_EXTI_ClearPending(&LCDTouchIRQ, EXTI_TRIGGER_RISING_FALLING);

    HAL_NVIC_ClearPendingIRQ(EXTI15_10_IRQn);
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

    // Clear IRQ bit again in case of errata
    WriteDataToTouchModule(STMPE811_INT_STA, clearIRQData);
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == GPIO_PIN_0) {
        RotateFallingBlock();
    }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM2) { // Check if TIM2 triggered the callback
        __HAL_TIM_DISABLE_IT(htim, TIM_IT_UPDATE); // Disable TIM2 interrupt
        elapsedTime++; // Increment elapsed time counter
        __HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE); // Reenable TIM2 interrupt
    }
    if (htim->Instance == TIM5) { // Check if TIM2 triggered the callback
        __HAL_TIM_DISABLE_IT(htim, TIM_IT_UPDATE); // Disable TIM5 interrupt
        Game();
        __HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE); // Reenable TIM5 interrupt
    }
}


#endif // TOUCH_INTERRUPT_ENABLED
```

A.2 Game.c

```
/*
 * Game.c
 *
 *  Created on: Dec 3, 2024
 *      Author: sammy
 */
#include "Game.h"

uint16_t blockMatrix[GRID_HEIGHT][GRID_WIDTH] = {0};

uint8_t fallingBlockMatrix[GRID_HEIGHT][GRID_WIDTH] = {0};
Block fallingBlock;
```

```
13
14  uint8_t elapsedTime = 0;
15  uint8_t singles = 0;
16  uint8_t doubles = 0;
17  uint8_t triples = 0;
18  uint8_t tetri = 0;
19
20  uint8_t currentFallingBlock = 0;
21  uint8_t nextBlock = 0;
22
23  static uint8_t blockBag[BAG_SIZE];
24  static uint8_t bagIndex = 0;
25  static uint8_t lastBlock = 255;
26
27  const Block tetrisBlocks[NUM_BLOCKS] = {
28      { // Square block
29          .shape = {
30              {1, 1, 0, 0},
31              {1, 1, 0, 0},
32              {0, 0, 0, 0},
33              {0, 0, 0, 0}
34          },
35          .width = 2,
36          .height = 2,
37          .color = LCD_COLOR_YELLOW
38      },
39      { // Line block
40          .shape = {
41              {1, 1, 1, 1},
42              {0, 0, 0, 0},
43              {0, 0, 0, 0},
44              {0, 0, 0, 0}
45          },
46          .width = 4,
47          .height = 1,
48          .color = LCD_COLOR_CYAN
49      },
50      { // T block
51          .shape = {
52              {0, 1, 0, 0},
53              {1, 1, 1, 0},
54              {0, 0, 0, 0},
55              {0, 0, 0, 0}
56          },
57          .width = 3,
58          .height = 2,
59          .color = LCD_COLOR_MAGENTA
60      },
61      { // L block
62          .shape = {
63              {1, 0, 0, 0},
64              {1, 1, 1, 0},
65              {0, 0, 0, 0},
66              {0, 0, 0, 0}
67          },
68          .width = 3,
69          .height = 2,
70          .color = LCD_COLOR_ORANGE
71      },
72      { // Reverse L block
73          .shape = {
74              {0, 0, 1, 0},
75              {1, 1, 1, 0},
76              {0, 0, 0, 0},
77              {0, 0, 0, 0}
78          },
79          .width = 3,
80          .height = 2,
```

```
        .color = LCD_COLOR_BLUE
    },
    { // S block
        .shape = {
            {0, 1, 1, 0},
            {1, 1, 0, 0},
            {0, 0, 0, 0},
            {0, 0, 0, 0}
        },
        .width = 3,
        .height = 2,
        .color = LCD_COLOR_GREEN
    },
    { // Z block
        .shape = {
            {1, 1, 0, 0},
            {0, 1, 1, 0},
            {0, 0, 0, 0},
            {0, 0, 0, 0}
        },
        .width = 3,
        .height = 2,
        .color = LCD_COLOR_RED
    }
};

void InitializeGame(void) {
    ShuffleBag();
    SpawnRandomBlock();

    // Setup Timers
    elapsedTime = 0;
    __HAL_TIM_SET_COUNTER(&htim2, 0);
    HAL_TIM_Base_Start_IT(&htim2);
    __HAL_TIM_SET_COUNTER(&htim5, 0);
    HAL_TIM_Base_Start_IT(&htim5);
}

void RenderGameScreen(void) {
    LCD_Clear(0, LCD_COLOR_BLACK);

    uint16_t gridBlockSize = LCD_PIXEL_HEIGHT / GRID_HEIGHT;

    // Render the grid
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            // Calculate top-left corner of the block
            uint16_t startX = x * gridBlockSize;
            uint16_t startY = y * gridBlockSize;

            // Draw grid lines (white border for all blocks)
            for (uint16_t i = 0; i < gridBlockSize; i++) {
                LCD_Draw_Pixel(startX + i, startY, LCD_COLOR_WHITE);                 // Top line
                LCD_Draw_Pixel(startX + i, startY + gridBlockSize - 1, LCD_COLOR_WHITE); // Bottom line
                LCD_Draw_Pixel(startX, startY + i, LCD_COLOR_WHITE);                 // Left line
                LCD_Draw_Pixel(startX + gridBlockSize - 1, startY + i, LCD_COLOR_WHITE); // Right line
            }

            // Render the static matrix (blockMatrix) with stored colors
            if (blockMatrix[y][x] != 0x0000) {
                for (uint16_t i = 1; i < gridBlockSize - 1; i++) {
                    for (uint16_t j = 1; j < gridBlockSize - 1; j++) {
                        LCD_Draw_Pixel(startX + i, startY + j, blockMatrix[y][x]);
                    }
                }
            }

            // Render the falling block (fallingBlockMatrix) with its color
```

```
            if (fallingBlockMatrix[y][x] == 1) {
                uint32_t color = fallingBlock.color;

                for (uint16_t i = 1; i < gridBlockSize - 1; i++) {
                    for (uint16_t j = 1; j < gridBlockSize - 1; j++) {
                        LCD_Draw_Pixel(startX + i, startY + j, color);
                    }
                }
            }
        }
    }
    DisplayTimer();
    RenderNextBlock();
}

void RenderNextBlock(void) {
    uint16_t nextBlockX = LCD_PIXEL_WIDTH - 80;
    uint16_t nextBlockY = 50;

    LCD_SetTextColor(LCD_COLOR_WHITE);
    LCD_SetFont(&Font16x24);
    LCD_DisplayString(nextBlockX, nextBlockY - 20, "Next:");
    LCD_Draw_Block(nextBlockX, nextBlockY, &tetrisBlocks[nextBlock]);
}

void UpdateFallingBlock(void) {
    for (int8_t y = GRID_HEIGHT - 1; y >= 0; y--) {
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (fallingBlockMatrix[y][x] == 1) {
                // Check collision with bottom or static blocks
                if (y == GRID_HEIGHT - 1 || blockMatrix[y + 1][x] != 0x000000) {
                    // Determine the color of the falling block
                    uint32_t blockColor = fallingBlock.color;

                    // Merge falling block into the static block matrix
                    for (uint8_t mergeY = 0; mergeY < GRID_HEIGHT; mergeY++) {
                        for (uint8_t mergeX = 0; mergeX < GRID_WIDTH; mergeX++) {
                            if (fallingBlockMatrix[mergeY][mergeX] == 1) {
                                blockMatrix[mergeY][mergeX] = blockColor; // Store the block's color
                                fallingBlockMatrix[mergeY][mergeX] = 0;  // Clear the falling block
                            }
                        }
                    }
                    ClearFullRows();
                    if (currentState != GAME_OVER)
                        SpawnRandomBlock();
                    return;
                }
            }
        }
    }

    // Move the falling block down
    for (int8_t y = GRID_HEIGHT - 1; y >= 0; y--) {
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (fallingBlockMatrix[y][x] == 1) {
                fallingBlockMatrix[y + 1][x] = 1;
                fallingBlockMatrix[y][x] = 0;
            }
        }
    }
}



void MoveFallingBlockLeft(void) {
    // Check if the block can move left
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
```

```
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (fallingBlockMatrix[y][x] == 1) {
                if (x == 0 || blockMatrix[y][x - 1] == 1) {
                    return; // Illegal move, do nothing
                }
            }
        }
    }

    // Move the block left
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (fallingBlockMatrix[y][x] == 1) {
                fallingBlockMatrix[y][x - 1] = 1;
                fallingBlockMatrix[y][x] = 0;
            }
        }
    }
}

void MoveFallingBlockRight(void) {
    // Check if the block can move right
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
        for (int8_t x = GRID_WIDTH - 1; x >= 0; x--) {
            if (fallingBlockMatrix[y][x] == 1) {
                if (x == GRID_WIDTH - 1 || blockMatrix[y][x + 1] == 1) {
                    return; // Illegal move, do nothing
                }
            }
        }
    }

    // Move the block right
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
        for (int8_t x = GRID_WIDTH - 1; x >= 0; x--) {
            if (fallingBlockMatrix[y][x] == 1) {
                fallingBlockMatrix[y][x + 1] = 1;
                fallingBlockMatrix[y][x] = 0;
            }
        }
    }
}

void RotateFallingBlock(void) {
    uint8_t tempMatrix[4][4] = {0};
    uint8_t blockWidth = fallingBlock.width;
    uint8_t blockHeight = fallingBlock.height;
    uint8_t topLeftX = GRID_WIDTH;
    uint8_t topLeftY = GRID_HEIGHT;

    // Find the minimum x and y coordinates of the falling block
    for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (fallingBlockMatrix[y][x] == 1) {
                if (x < topLeftX) topLeftX = x;
                if (y < topLeftY) topLeftY = y;
            }
        }
    }

    // Extract the current block into tempMatrix
    for (uint8_t y = 0; y < BLOCK_SIZE; y++) {
        for (uint8_t x = 0; x < BLOCK_SIZE; x++) {
            tempMatrix[y][x] = fallingBlockMatrix[topLeftY + y][topLeftX + x];
        }
    }

    // Transpose the matrix
```

```
285     for (uint8_t i = 0; i < BLOCK_SIZE; i++) {
286         for (uint8_t j = i + 1; j < BLOCK_SIZE; j++) {
287             uint8_t temp = tempMatrix[i][j];
288             tempMatrix[i][j] = tempMatrix[j][i];
289             tempMatrix[j][i] = temp;
290         }
291     }
292 
293     // Reverse each row
294     for (uint8_t i = 0; i < BLOCK_SIZE; i++) {
295         uint8_t start = 0, end = BLOCK_SIZE-1;
296         while (start < end) {
297             uint8_t temp = tempMatrix[i][start];
298             tempMatrix[i][start] = tempMatrix[i][end];
299             tempMatrix[i][end] = temp;
300             start++;
301             end--;
302         }
303     }
304 
305     // Shift the tempMatrix left by the calculated amount for every row
306     uint8_t shiftAmount = (fallingBlock.width < fallingBlock.height) ? 1 : 2;
307     for (uint8_t i = 0; i < BLOCK_SIZE; i++) {
308         for (uint8_t j = shiftAmount; j < BLOCK_SIZE; j++)
309             tempMatrix[i][j - shiftAmount] = tempMatrix[i][j];
310         for (uint8_t j = BLOCK_SIZE - shiftAmount; j < BLOCK_SIZE; j++)
311             tempMatrix[i][j] = 0;
312     }
313 
314     // Check for boundary collisions or overlaps
315     for (uint8_t y = 0; y < BLOCK_SIZE; y++) {
316         for (uint8_t x = 0; x < BLOCK_SIZE; x++) {
317             if (tempMatrix[y][x] == 1) {
318                 uint8_t gridX = topLeftX + x;
319                 uint8_t gridY = topLeftY + y;
320 
321                 if (gridX >= GRID_WIDTH || gridY >= GRID_HEIGHT || blockMatrix[gridY][gridX] != 0x0000) {
322                     return;
323                 }
324             }
325         }
326     }
327 
328     // Clear the current falling block from the grid
329     for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
330         for (uint8_t x = 0; x < GRID_WIDTH; x++) {
331             if (fallingBlockMatrix[y][x] == 1) {
332                 fallingBlockMatrix[y][x] = 0;
333             }
334         }
335     }
336 
337     // Apply rotated block back to the falling block matrix
338     for (uint8_t y = 0; y < BLOCK_SIZE; y++) {
339         for (uint8_t x = 0; x < BLOCK_SIZE; x++) {
340             if (tempMatrix[y][x] == 1) {
341                 fallingBlockMatrix[topLeftY + y][topLeftX + x] = 1;
342             }
343         }
344     }
345 
346     // Update the falling block's dimensions for multiple rotation purposes
347     fallingBlock.width = blockHeight;
348     fallingBlock.height = blockWidth;
349 }
350 
351 void ShuffleBag(void) {
352     // Fill the bag with block indices
```

```
    for (uint8_t i = 0; i < BAG_SIZE; i++) {
        blockBag[i] = i;
    }

    // Shuffle the bag
    for (uint8_t i = BAG_SIZE - 1; i > 0; i--) {
        uint32_t rngValue = 0;
        HAL_RNG_GenerateRandomNumber(&hrng, &rngValue);
        uint8_t j = rngValue % (i + 1);
        uint8_t temp = blockBag[i];
        blockBag[i] = blockBag[j];
        blockBag[j] = temp;
    }
}

void FormatTimerString(char *timerString, uint32_t minutes, uint32_t seconds) {
    timerString[0] = '0' + (minutes / 10);  // First digit of minutes
    timerString[1] = '0' + (minutes % 10);  // Second digit of minutes
    timerString[2] = ':';                   // Colon separator
    timerString[3] = '0' + (seconds / 10);  // First digit of seconds
    timerString[4] = '0' + (seconds % 10);  // Second digit of seconds
    timerString[5] = '\0';                  // Null terminator
}

// Function to display the timer on the screen
void DisplayTimer(void) {
    char timerString[TIMER_NUMCHARS];

    // Format elapsedTime into a string (e.g., "00:00")
    uint32_t minutes = elapsedTime / 60;
    uint32_t seconds = elapsedTime % 60;
    FormatTimerString(timerString, minutes, seconds);

    // Calculate position to render the timer (top-right)
    uint16_t timerX = LCD_PIXEL_WIDTH - (CHAR_WIDTH * (TIMER_NUMCHARS-1)); // Adjust based on character width (5 chars)
    uint16_t timerY = Y_OFF;

    // Display the timer text in white
    LCD_SetTextColor(LCD_COLOR_WHITE);
    LCD_SetFont(&Font16x24);
    uint16_t currentX = timerX; // Track position for each character
    for (const char *p = timerString; *p != '\0'; p++) {
        LCD_DisplayChar(currentX, timerY, *p);
        currentX += CHAR_WIDTH; // Advance by character width
    }
}

void ClearFullRows(void) {
    uint8_t clearedRows = 0;

    for (int8_t y = GRID_HEIGHT - 1; y >= 0; y--) {
        bool isFullRow = true;

        // Check if the row is full
        for (uint8_t x = 0; x < GRID_WIDTH; x++) {
            if (blockMatrix[y][x] == 0x0000) {
                isFullRow = false;
                break;
            }
        }

        if (isFullRow) {
            clearedRows++;

            // Clear the row
            for (uint8_t x = 0; x < GRID_WIDTH; x++) {
                blockMatrix[y][x] = 0x0000;
            }
```

```
421
422             // Shift rows above down
423             for (int8_t row = y; row > 0; row--) {
424                 for (uint8_t col = 0; col < GRID_WIDTH; col++) {
425                     blockMatrix[row][col] = blockMatrix[row - 1][col];
426                 }
427             }
428
429             // Clear the top row (after shift)
430             for (uint8_t x = 0; x < GRID_WIDTH; x++) {
431                 blockMatrix[0][x] = 0x0000;
432             }
433
434             // Adjust row index to recheck the current row after shifting
435             y++;
436         }
437     }
438
439     // Update scoring based on the number of cleared rows
440     switch (clearedRows) {
441         case 1:
442             singles++;
443             break;
444         case 2:
445             doubles++;
446             break;
447         case 3:
448             triples++;
449             break;
450         case 4:
451             tetri++;
452             break;
453         default:
454             break;
455     }
456 }
457
458
459 void SpawnFallingBlock(uint8_t blockIndex) {
460     // Clear the current falling block matrix
461     for (uint8_t y = 0; y < GRID_HEIGHT; y++) {
462         for (uint8_t x = 0; x < GRID_WIDTH; x++) {
463             fallingBlockMatrix[y][x] = 0;
464         }
465     }
466
467     fallingBlock = tetrisBlocks[blockIndex];
468
469     // Place the block at the top center of the grid
470     uint8_t startX = (GRID_WIDTH - fallingBlock.width) / 2;
471     for (uint8_t y = 0; y < fallingBlock.height; y++) {
472         for (uint8_t x = 0; x < fallingBlock.width; x++) {
473             if (fallingBlock.shape[y][x]) {
474                 fallingBlockMatrix[y][startX + x] = fallingBlock.shape[y][x];
475             }
476         }
477     }
478 }
479
480 void SpawnRandomBlock(void) {
481     if (bagIndex >= BAG_SIZE) {
482         ShuffleBag();
483         bagIndex = 0;
484     }
485
486     currentFallingBlock = nextBlock;
487     nextBlock = blockBag[bagIndex++];
488
```

```
    if (nextBlock == lastBlock && BAG_SIZE > 1) {
        uint8_t swapIndex = (bagIndex < BAG_SIZE) ? bagIndex : 0;
        uint8_t temp = nextBlock;
        nextBlock = blockBag[swapIndex];
        blockBag[swapIndex] = temp;
    }

    lastBlock = nextBlock;
    SpawnFallingBlock(currentFallingBlock);
}
```

A.3 GameOver.c

```
/*
 * GameOver.c
 *
 *  Created on: Dec 3, 2024
 *      Author: sammy
 */

#include <stdio.h>
#include "GameOver.h"
#include "Game.h"

void CheckGameOver(void) {
    // Iterate through the top row of the matrix to check for any filled blocks
    for (uint8_t x = 0; x < GRID_WIDTH; x++) {
        if (blockMatrix[0][x] != 0x0000) {
            currentState = GAME_OVER;
            HAL_TIM_Base_Stop_IT(&htim2);
            RenderGameOverScreen();
            return;
        }
    }
}

void RenderGameOverScreen(void) {
    char buffer[32];

    LCD_Clear(0, LCD_COLOR_BLACK);
    LCD_SetTextColor(LCD_COLOR_WHITE);
    LCD_SetFont(&Font16x24);

    uint16_t textX = (LCD_PIXEL_WIDTH / 2) - (4 * 16);
    uint16_t textY = LCD_PIXEL_HEIGHT / 6;
    LCD_DisplayString(textX, textY, "GAME OVER");

    // Display elapsed time
    textX = STAT_X;
    textY += BUFFER_Y;
    sprintf(buffer, "Time: %02u:%02u", elapsedTime / 60, elapsedTime % 60);
    LCD_DisplayString(textX, textY, buffer);

    // Display stats: singles, doubles, triples, tetrises
    textY += BUFFER_Y;
    sprintf(buffer, "Singles: %u", singles);
    LCD_DisplayString(textX, textY, buffer);

    textY += BUFFER_Y;
    sprintf(buffer, "Doubles: %u", doubles);
    LCD_DisplayString(textX, textY, buffer);

    textY += BUFFER_Y;
    sprintf(buffer, "Triples: %u", triples);
    LCD_DisplayString(textX, textY, buffer);

```

```
    textY += BUFFER_Y;
    sprintf(buffer, "Tetrises: %u", tetri);
    LCD_DisplayString(textX, textY, buffer);
}
```

A.4 MainMenu.c

```
/*
 * MainMenu.c
 *
 *  Created on: Dec 3, 2024
 *      Author: sammy
 */

#include <stdbool.h>
#include "MainMenu.h"
#include "Game.h"

// Constants for the play button
static const uint16_t playButtonX = 120;
static const uint16_t playButtonY = 160;
static const uint16_t playButtonSize = 20;

// Constants for block placement
static const uint16_t buffer = 5; // Buffer size in pixels

bool IsOverlap(uint16_t Xpos, uint16_t Ypos, const Block *block, uint16_t positions[][2], uint8_t numPlacedBlocks) {
    uint16_t blockWidth = block->width * RENDER_BLOCK_SIZE + buffer;
    uint16_t blockHeight = block->height * RENDER_BLOCK_SIZE + buffer;

    // Check overlap with previously placed blocks
    for (uint8_t i = 0; i < numPlacedBlocks; i++) {
        uint16_t otherX = positions[i][0];
        uint16_t otherY = positions[i][1];
        uint16_t otherWidth = tetrisBlocks[i].width * RENDER_BLOCK_SIZE + buffer;
        uint16_t otherHeight = tetrisBlocks[i].height * RENDER_BLOCK_SIZE + buffer;

        if (!(Xpos + blockWidth <= otherX || // No overlap to the left
              Xpos >= otherX + otherWidth || // No overlap to the right
              Ypos + blockHeight <= otherY || // No overlap above
              Ypos >= otherY + otherHeight)) { // No overlap below
            return true; // Overlap detected
        }
    }

    // Check overlap with the play button
    uint16_t playButtonLeft = playButtonX - playButtonSize - buffer;
    uint16_t playButtonRight = playButtonX + playButtonSize + buffer;
    uint16_t playButtonTop = playButtonY - playButtonSize - buffer;
    uint16_t playButtonBottom = playButtonY + playButtonSize + buffer;

    if (!(Xpos + blockWidth <= playButtonLeft || // No overlap to the left of the button
          Xpos >= playButtonRight ||             // No overlap to the right of the button
          Ypos + blockHeight <= playButtonTop || // No overlap above the button
          Ypos >= playButtonBottom)) {           // No overlap below the button
        return true; // Overlap with play button detected
    }

    return false; // No overlap
}

void PlaceBlock(uint16_t *Xpos, uint16_t *Ypos, const Block *block, uint16_t positions[][2], uint8_t numPlacedBlocks) {
    do {
        *Xpos = rand() % (LCD_PIXEL_WIDTH - block->width * RENDER_BLOCK_SIZE);
```

```
58          *Ypos = rand() % (LCD_PIXEL_HEIGHT - block->height * RENDER_BLOCK_SIZE);
59      } while (IsOverlap(*Xpos, *Ypos, block, positions, numPlacedBlocks));
60  }
61
62  void DrawPlayButton(void) {
63      for (int16_t y = -playButtonSize; y <= playButtonSize; y++) {
64          for (int16_t x = -playButtonSize; x <= playButtonSize; x++) {
65              if (x < 0 && abs(x) >= abs(y)) {
66                  LCD_Draw_Pixel(playButtonX + x, playButtonY + y, LCD_COLOR_WHITE);
67              }
68          }
69      }
70  }
71
72  void DisplayMenu(void) {
73      // Clear the screen with black background
74      LCD_Clear(0, LCD_COLOR_BLACK);
75
76      // Array to store positions of placed blocks
77      uint16_t positions[NUM_BLOCKS][2] = {0};
78
79      // Display all Tetris blocks scattered on the screen
80      for (uint8_t i = 0; i < NUM_BLOCKS; i++) {
81          uint16_t randX, randY;
82          PlaceBlock(&randX, &randY, &tetrisBlocks[i], positions, i);
83          positions[i][0] = randX;
84          positions[i][1] = randY;
85          LCD_Draw_Block(randX, randY, &tetrisBlocks[i]);
86      }
87
88      // Draw the "Play" button
89      DrawPlayButton();
90  }
```